A Sound Modeling and Synthesis System Designed for Maximum Usability

Lonce Wyse

Institute for Infocomm Research, Singapore
email: lonce@zwhome.org, www.zwhome.org/~lonce

Abstract

A synthesis and sound-modeling system is introduced. The design philosophy is to be "good enough most of the time" in an extremely wide variety of real-world scenarios at the possible expense of being the best in any one particular aspect - including speed and reliability. Sound design and synthesis system design goals and decisions are discussed, and we consider the appropriateness of using pure Java as an implementation language for such systems.

1 Design Constraints

There are a plethora of software synthesizers and sound development environments to choose between today, and each has its own strengths and weaknesses in meeting the many different design goals for a sound design and synthesis (SDS) system. Often the design goals support conflicting decisions in implementation. For example, the goal of expressivity may support the invention of a new language oriented specifically toward sonic and musical tasks, while the goal of learnability may be better addressed by using a language already familiar to many people. We have developed an SDS system, called Asound, with maximum usability being the primary design criteria, written in pure Java.

Common design objectives for SDS systems revolve around the following issues:

- Speed – for real-time performance and shortest delay between input audio or control signals and audio output.
- Reliability – dependable delivery of uninterrupted audio to the output device.
- Expressivity– the ability for the code to be written in the musical and/or sonic terms in which composers and sound designers think.
- Power – the range of available tools (e.g. a large number of *unit generators*, or sound producing and filtering routines)
- Learnability – the system needs to be as usable as possible by musicians and sound designers, even if they are not expert programmers. This is part of the motivation for graphical interfaces such as MAX (Puckette, 1991).
- Fast development times – The time it takes to develop bug free sound models should be minimized.
- Development support – a good integrated development and debugging environment.
- Usability in education – a combination of expressivity and learnability.
- Ubiquity – the system should be inexpensive and not require special purpose hardware and software.
- Support for complexity – the ability to write richly structured sound models in readable code and to "hide" complexity in functions and objects.
- Absence of musical and sonic structure biases – the system should bias the user as little as possible as to the genre of music or sonic style. For example, it should be possible to integrate algorithms for event and sound generation.
- Extensibilty – Sound developers and users need to be able to extend the capabilities of an SDS system since no particular one will ever meet all the needs of designers and composers. Furthermore, sound modeling is an active field, and an SDS system needs to be designed to grow as new needs and possibilities arise.
- Cross-platform potential - It should be possible to develop sound and musical objects on whichever platform is most convenient for the composer or developer, and then run on most others.
- Integrability – Sound objects should be available to a maximum number of other applications environments – to code written in languages other than the one used to develop the sound object, or to external MIDI controllers. They should enable control from and if necessary, return audio to other applications such as sequencers, audio editors, graphical applications, games, and multimedia development environments such as Macromedia Director & Flash, 3DS Max.

- Maintainabilty – easy to upgrade and modify with the minimum amount of effort. This argues against graphical interfaces.
- Small bandwidth requirements – a major concern when downloading the system or sound models is required for such environments as applets, interactive Macromedia applications, and online or downloadable games.
- Low security risk - client computers must be safe from the possibility of downloading malicious code.

2 Meeting Design Requirements with Java

By building the SDS system in pure Java, many of the above requirements are automatically met. The core of the system that address the sound design requirements *per s*e are addressed below. Addressing ubiquity and learnability, Java is a free, commonly used and widely taught language. For the many thousands of programmers, only the specific library of classes for sound need be learned. For development support, there are commercially supported online manuals, and extensive tutorial material available across the Web. There are free and commercial IDE's (integrated development environments) that support object viewing, graphical interface design, and state-of-the-art debugging tools. It is considerably faster to develop code in Java than in C or C++, something we now have the luxury to consider with the execution speeds that modern desktop computers are achieving.

Having the full power of a general-purpose language is important for several reasons. It helps circumvent biases built in to the sound and musical construction process that are hard or impossible to work around in more constrained task-specific languages. It permits elegant coding style and the possibility of developing with manageable complexity.

Both the core ASound system and Sound Models run without modification on any platform with a modern Java VM. By creating a core set of sound and musical classes for programming, and not providing a graphical coding environment, the system is both easy to use and easy to maintain.

Design pressures on SDS systems that have been growing in importance have to do with the growing need to send applications, plug-ins and/or sound models over the network. The core ASound system is under 60 Kb, and sound models that don't require audio file resources are typically from 2 to 5 Kb. These are manageable numbers for even the most limited memory devices, and make download time negligible.

A Java-based system also poses no security threat to clients. ASound sound models are executable Java byte code (not simply parameters for predefined synthesizers). For a core engine it is at least feasible to require end users to grant a one-time security certificate, but many different individual sound models are used in typical applications and can they come from many different developers and vendors. This would create an insurmountable security problem for sound models written in C, for example.

3 Meeting Design Constraints with Modular System Design

The central design unit of the system is the Sound Model. A standard interface affords Play(), Stop() and Release() events, and continuous parameter access (setting and getting) in both natural units, and in normalized floating point [0,1] units. Rather than have sound model-specific and parameter-specific methods for control, the interface methods for control use a parameter index retrieved from the sound using the parameter string name so that the interface methods are the same for all sound models.

Sound Models return audio with a call to a Generate() method that takes an empty buffer and a requested number of samples to fill it with as arguments.

A separate object, the SoundManager, controls a back-end output engine with an audio buffer and a timer. The SoundManager manages a list of sound models by periodically calling their Generate() methods, and summing the results into the audio output buffer. This backend is entirely separate from the sound model and need not be used at all. An application can manage the Generate() calls itself. This situation arises, for example, if an application has access to a machine-specific buffer architecture (e.g. Creative EAX buffers on Wintel environments). Another example of not using the SoundManager synthesis backend is a non-realtime application for creating audio files from a musical score. Such an application calls sound model Generate() methods at whatever (possibly irregular) intervals are appropriate for the time stamps of the events in the score and concatenates the returned audio to the end of a file.

Input control is similarly separate from sound models. For example, sound models never contain MIDI specific (or even more confounding, graphical interface) code. An entirely separate MIDI synthesizer application manages MIDI input and mappings to a list of sound models, and is used as a recipient for messages streaming from a commercial midi sequencer application. By separating the backend timer/buffer engine and the front-end control systems from the sound models, the sound models are clean, small, and useful in a maximum variety of contexts.

4 Sound Model Design

A library of classes for standard structures and unit generators (e.g. oscillators, filters) is used to build a sound model. A key feature of the system is that event generation and audio generation are supported on an equal footing. The Sound Model Generate() method basically calls two methods in sequence; GenerateEvents() and GenerateAudio(). Both perform their computation up to a specified time corresponding to the length of the buffer fill requested from Generate(). The event generator uses the standard model interface (parameter changes, starts, stops, releases as described above) with an additional time stamp to send events to a "submodel" which puts them on a queue that is managed with sample accuracy. If the model uses event generation for a submodel, then its audio generator typically takes responsibility for getting the audio from the submodel by calling the submodel Generate() method. The structure is shown in Figure 1.

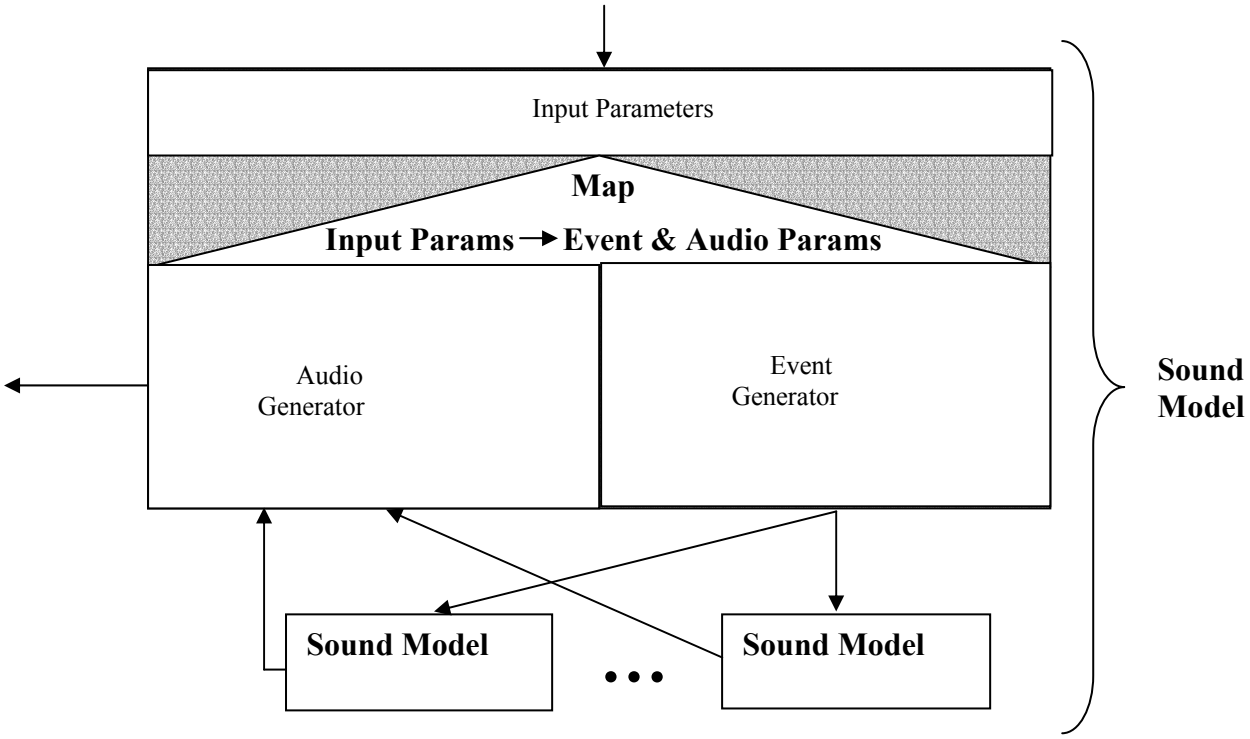

Figure 1. The basic (recursive) model structure.

4 Common Sound Model Structures

Single-event Audio Generation – this is the standard parameterized synthesizer paradigm that generates a single "event" in response to a "play" command and offers realtime parametric control. When the Generate method on such a model is called, the Event Generator does nothing, and the Audio Generator return the audio. The Audio Generator synthesis algorithm may have many dimensions of control to which the exposed model parameters are mapped.

Single synthesis algorithm with event pattern control – uses the model-with-a-model structure as shown Figure 1, but with a single submodel. Often, parameters of the audio-only generating submodel are exposed without modification by the top level model, and additional parameters are exposed to control the event pattern generation. Such a structure is suitable for, as an example, an engine sound where the audio-only submodel just generates a burst of noise, and the event generating "wrapper" manages piston firings. Parameters for the noise burst submodel, such as filter or envelope characteristics, can be exposed by the wrapper model and passed through to the submodel. This structure is so typical, that an ASound core class provides just that capability. The Sound Model developer only has to provide the synthesis submodel, create the control parameters and mappings, and override the event generating method.

Multiple submodels, audio postprocessing – The top-level model generates events and parameter patterns that coordinate the sound synthesis across many elementary component models. The

GenerateAudio() method at the top level retrieves the submodel audio and can then post-process (add reverb, filtering, panning, or any arbitrary processing or synthesis code) before returning the buffer to the original caller. This hierarchical method of building up models by coordinating events and control parameters across simpler models has proven to be a very powerful, efficient and intuitive way to construct rich and complex sound and musical objects.

In addition to the audio and event generator methods, sound design typically involves determining what parameters to expose. This is done by creating a parameter object with a string name, minimum, maximum and default values, and adding it to the sound in an initialization method. Once this is done, the parameter is automatically available to the sound model interface as described above.

Several other methods are often overridden by a particular sound model:

- OnInitialize() – executed once only after a sound is loaded and before it is used,
- OnParameterUpdate() – This is where any mapping is done between parameters exposed by the model and controls for the audio and event generators can be done,
- OnPlay() – executed when a play event is received and before audio generation begins,
- OnRelease() – overridden if the sound should enter a release segment rather than stop generating sound immediately,
- OnStop() – executed when the sound actually stops generating audio,
- OnK() – executed periodically at a per-sound settable rate assumed to be slower than the sampling rate.

The expressivity design goal is thus addressed by appropriately hiding the bookkeeping of event queue management, providing methods that correspond to logical components of a sound model, and the modular separation of synthesis and specific input controls form the sound modeling code. With a surprisingly small number of sound & music oriented classes, coding in Java can be as expressive as coding in a special purpose language such as SAOL . At the same time, no new language constructs need be learned (sometimes considered a hurdle to the otherwise powerful and expressive SuperCollider (McCartney 2002)), no new integrated development environments need be developed (simply lacking in most special-purpose languages), and the full power of a general purpose language is still available as needed.

5 Other design issues

Sound model file formats in ASound are essentially jar files that include the sound model and any subsound model classes, any other classes not defined in ASound, and any audio resources the models might use. The core system includes a special Java class loader to manage the format, but since it is essentially a jar file, the Java language itself provides most of the tools for building the class loader. The jar format includes a certain amount of compression, and the sound model file format can be entirely self-contained, with no dependencies beyond the core system.

The core system comes with only a small set of unit generators; those that are very commonly used such as a table reader, oscillators, and filters. This helps keep the core system small by not including rarely used classes, but comes at the minor cost of possibly having to download a sound-specific unit generator class (for example) twice if two different sounds in a single application make use of it.

Finally, sound developers will find the system is easy to extend with additional classes by simply compiling them, jarring them together and putting the jar file in the Java classpath. There is no need to change any "glue" code, or recompile any part of the core system as there is in some other special purpose synthesis languages. ASound can also be called from C programs via a static library wrapper written with JNI (Java Native Interface). The wrapper code is technical and tedious to write, and compiled to a machine-specific form, but such a burden would not fall on the music and sound developers who do their sound-oriented development once and for all.

One of the most well-known music languages is Csound which is in the MUSIC N (Mathews, 1969) lineage of languages specifically designed for audio processing and synthesis. Csound is fast, has a vast collection of unit generators available, is free and used widely as an educational tool and has a broad user base. However, it is quite limited compared with modern high-level computer languages like C and Java in areas such as data structures, control structures, and integrated development environments for supporting coding and debugging.

Jsyn (Burk) combines the best of both worlds by having native method backends, while sound and music developers work in Java. Other than the core system that comes from a single vender, only Java code is shared between developers and users of sounds. This solves the security issues associated with having to download native code for sounds, and provides a speed advantage. However, with modern Java environments, the speed advantage this architecture would have provided a few years ago isn't as significant today. The worst of both worlds comes with Java/native hybrid systems, too. They are more difficult to maintain and to make cross platform than are pure Java systems, and are still subject to realtime disruption from Java garbage collection.

There are, of course, several reasons why pure Java SDS systems are not wide spread. One drawback is speed (we typically see execution speeds about 1.4

times that of comparable native code), and another is the unpredictable garbage collection in Java that makes such a system simply unusable in some musical contexts. Furthermore, the current implementation of the Java millisecond timer (as of Java SDK v1.4.2) is still quite poor on most Windows platforms, accurate only to within about 60 ms. It is far more reliable on other platforms (Linux and Mac), and a temporary workaround (in native Windows code) provides timer accuracy to within about 1 ms. Finally, it is still unclear whether Microsoft will finally provide a standard Java virtual machine with their operating systems. Currently their customers must download and install one themselves, an impediment to cross-platform development in general.

For non-realtime music generation from complex models and scores, the speed and realtime reliability are not a factor, making the design constraints that the ASound system best addresses entirely appropriate. For sounds embedded in realtime games and applications, the occasional hiccup that random garbage collection or poor timer implementations can cause have to be weighed against the value of the benefits of security, size, platform, development support, etc. For professional realtime music performance, the system cannot compare to more specialized systems – but the demands of the network environment, faster computers, just-in-time compilers, and ever better JVM's are making a well-designed Java class library SDS system a very viable option for most of the people most of the time.

References

Burke, Phil. Jsyn; Audio Software Synthesis API and Plugins for Java. http://www.softsynth.com/jsyn/

Mathews, M., 1969. *Technology of Computer Music*. Cambridge, Mass., M.I.T. Press.

McCartney, James, 2002. Rethinking the Computer Music Language: SuperColider. *Computer Music Journal* **26**(4).

Puckette, Miller, 1991. Combining event and signal processing in the MAX graphical programming environment," *Computer Music Journal,* **15**(3):68 - 77.